Radix Sort

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort Algorithm

The key idea behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

How does Radix Sort Algorithm work?

To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:

Step 1: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.

Step 2: Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place.

Sorting based on the unit place:

Perform counting sort on the array based on the unit place digits.
The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66].

Step 3: Sort the elements based on the tens place digits.

Sorting based on the tens place:

Perform counting sort on the array based on the tens place digits.
The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].

Step 4: Sort the elements based on the hundreds place digits.

Sorting based on the hundreds place:

Perform counting sort on the array based on the hundreds place digits.
The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].

Step 5: The array is now sorted in ascending order.

The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].

Counting Sort

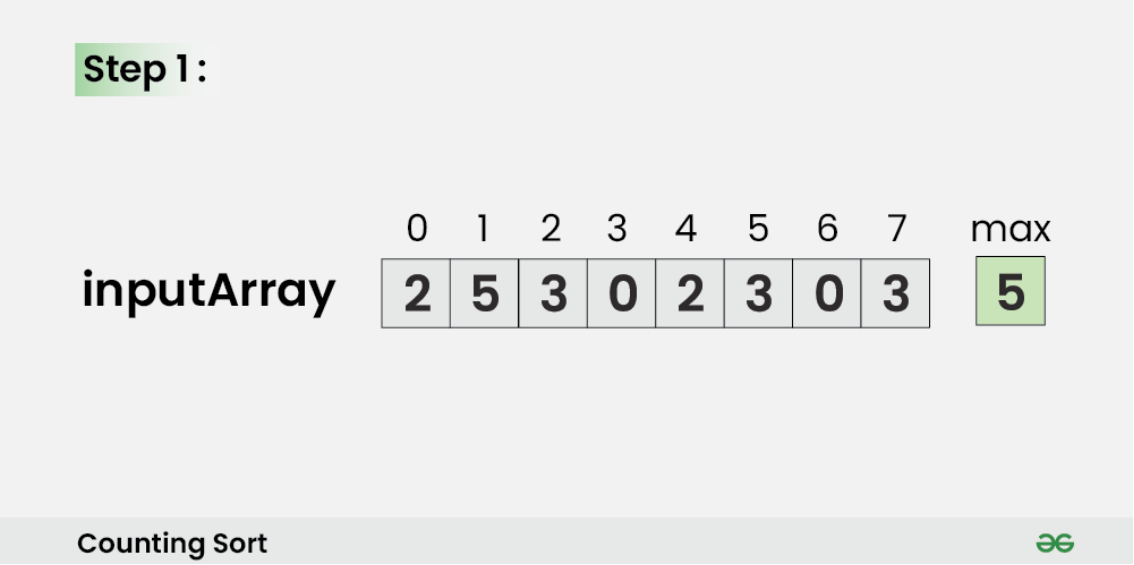
What is Counting Sort?

Counting Sort is a **non-comparison-based** sorting algorithm that works well when there is limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind **Counting Sort** is to count the **frequency** of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

How does Counting Sort Algorithm work?

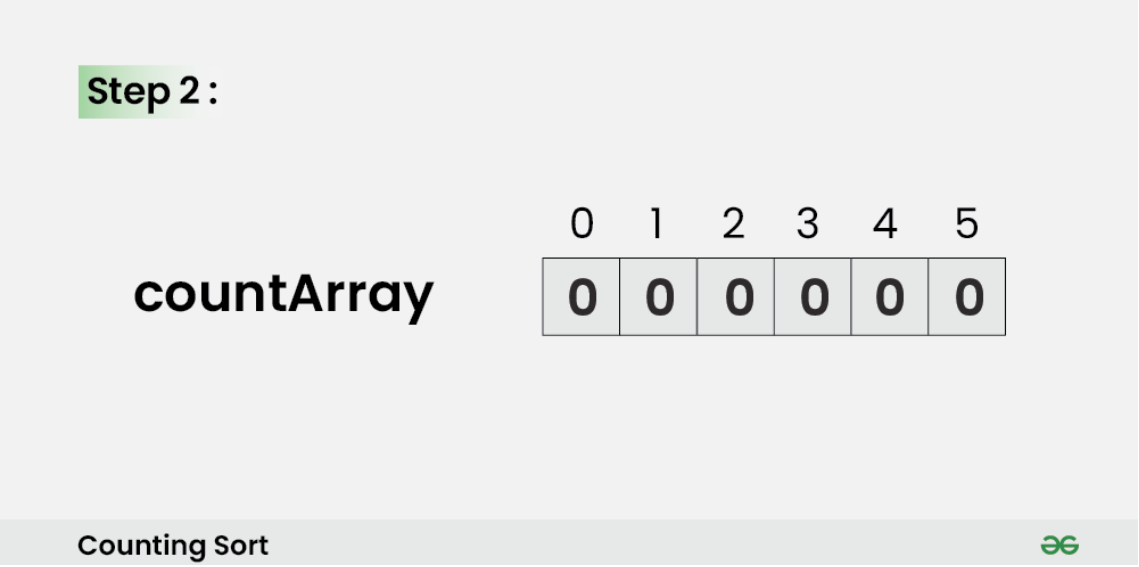
Step1 :

- Find out the **maximum** element from the given array.



Step 2:

- Initialize a **countArray[]** of length **max+1** with all elements as **0**. This array will be used for storing the occurrences of the elements of the input array.



Step 3:

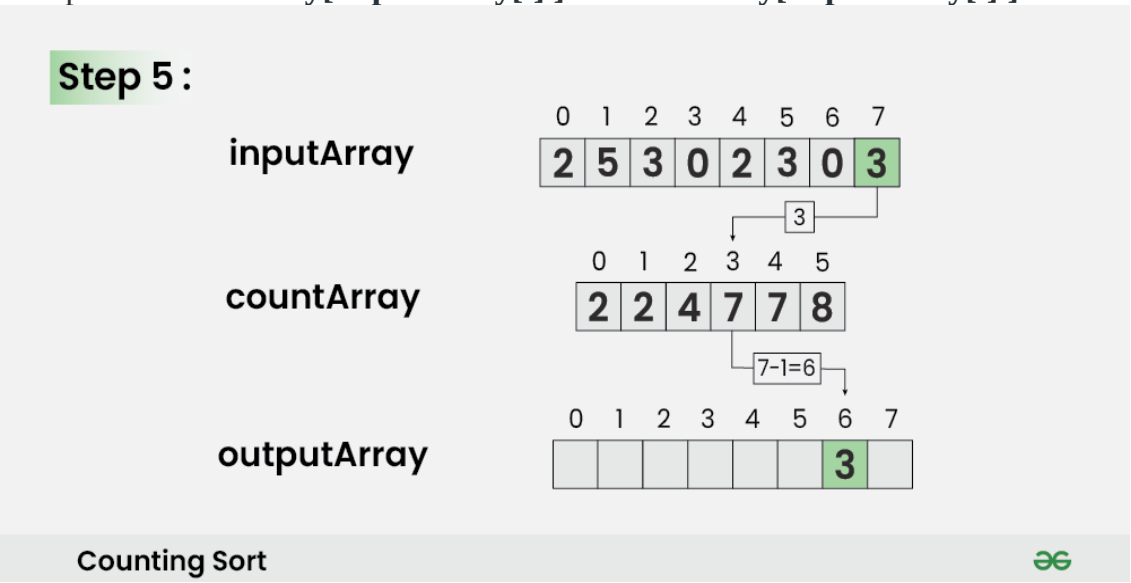
- In the **countArray[]**, store the count of each unique element of the input array at their respective indices.
- **For Example:** The count of element **2** in the input array is **2.** So, store **2** at index **2** in the **countArray[]**. Similarly, the count of element **5** in the input array is **1**, hence store **1** at index **5** in the **countArray[]**.

Step 4:

- Store the **cumulative sum** or **prefix sum** of the elements of the **countArray[]** by doing **countArray[i] = countArray[i – 1] + countArray[i].** This will help in placing the elements of the input array at the correct index in the output array.

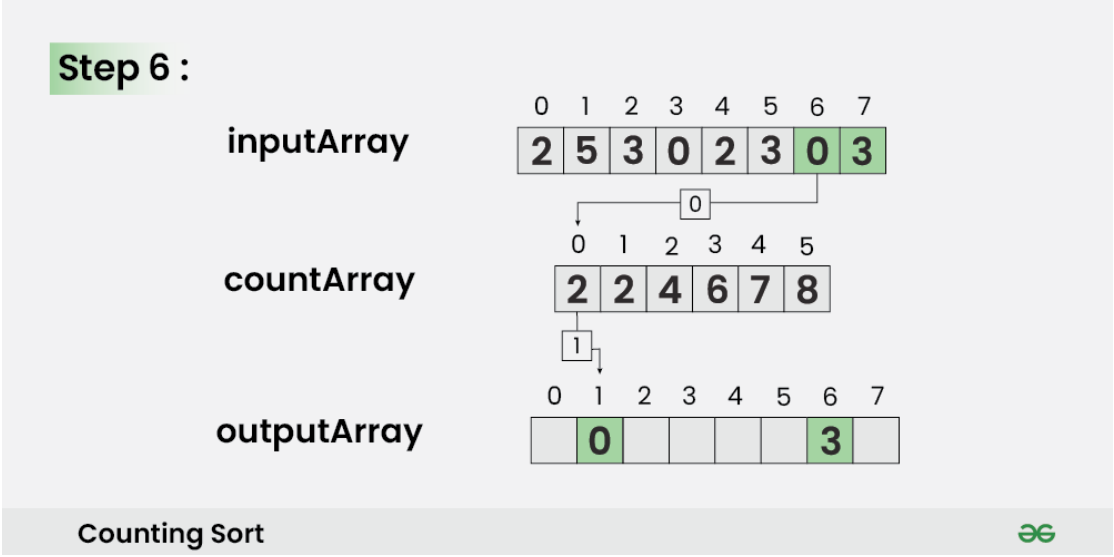
Step 5:

- Iterate from end of the input array and because traversing input array from end preserves the order of equal elements, which eventually makes this sorting algorithm **stable**.
- Update **outputArray[countArray[inputArray[i]] – 1] = inputArray[i]**. Also, update **countArray[inputArray[i]] = countArray[inputArray[i]]**–.



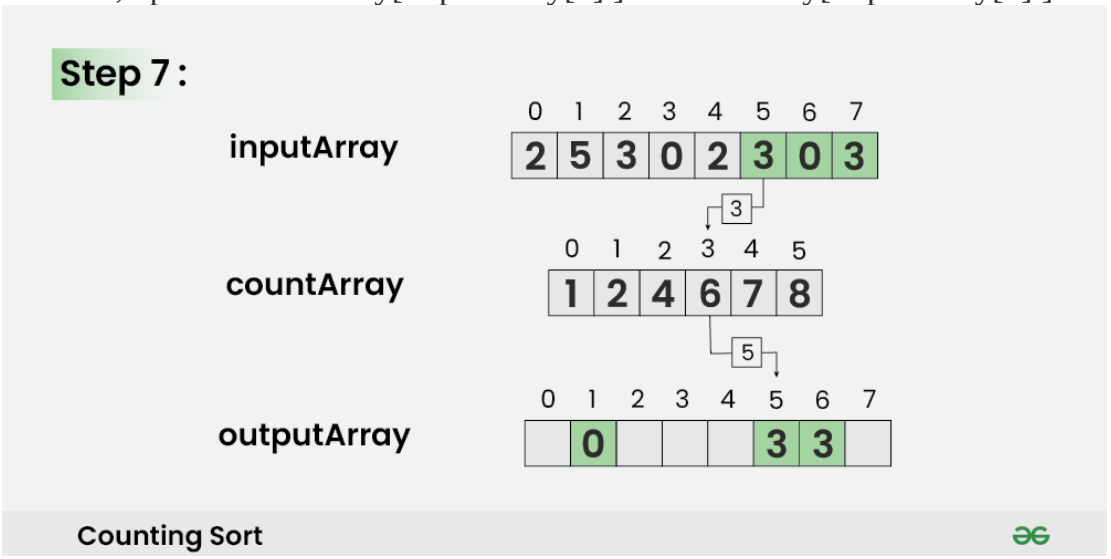
Step 6:

- **For i = 6**, Update outputArray[countArray[inputArray[6]] – 1] = inputArray[6]. Also, update countArray[inputArray[6]] = countArray[inputArray[6]]–.



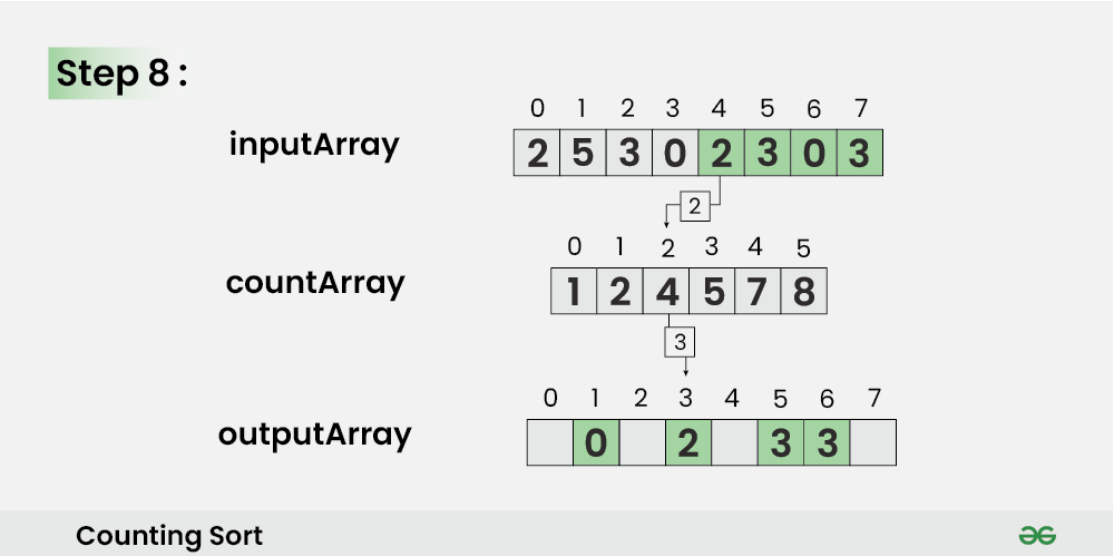
Step 7:

- **For i = 5**, Update outputArray[countArray[inputArray[5]] – 1] = inputArray[5]. Also, update countArray[inputArray[5]] = countArray[inputArray[5]]–.



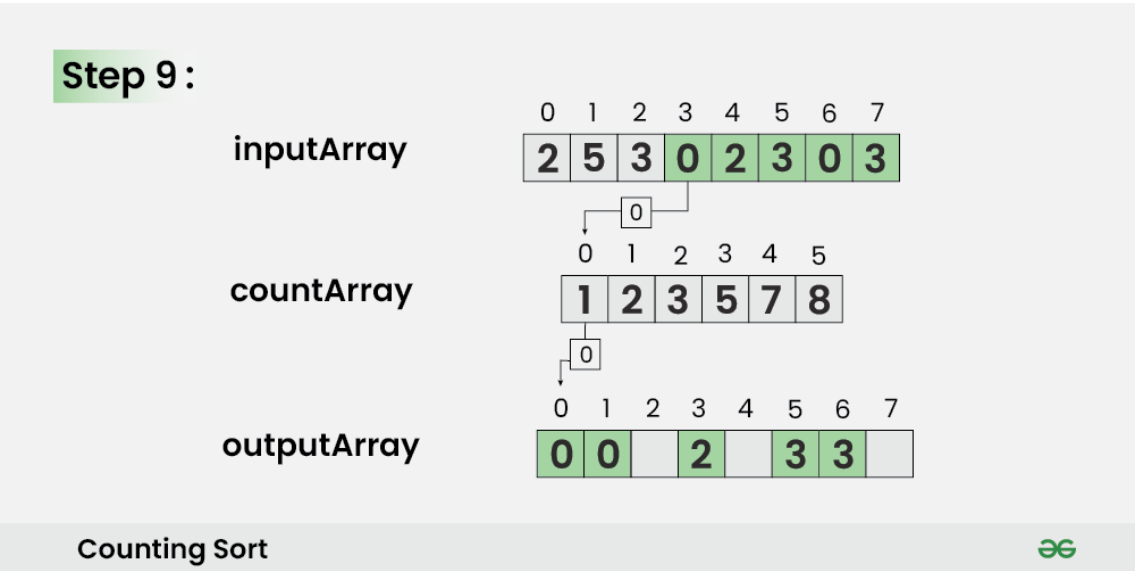
Step 8:

- **For i = 4**, Update outputArray[countArray[inputArray[4]] – 1] = inputArray[4]. Also, update countArray[inputArray[4]] = countArray[inputArray[4]]–.



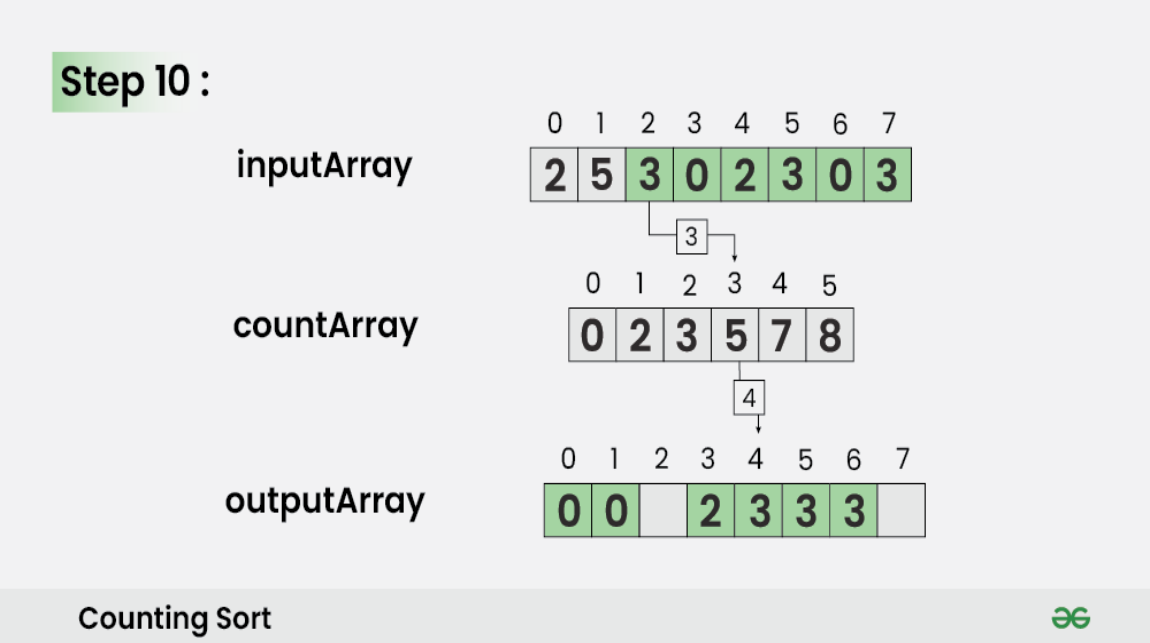
Step 9:

For i = 3, Update outputArray[countArray[inputArray[3]] – 1] = inputArray[3]. Also, update countArray[inputArray[3]] = countArray[inputArray[3]]–.



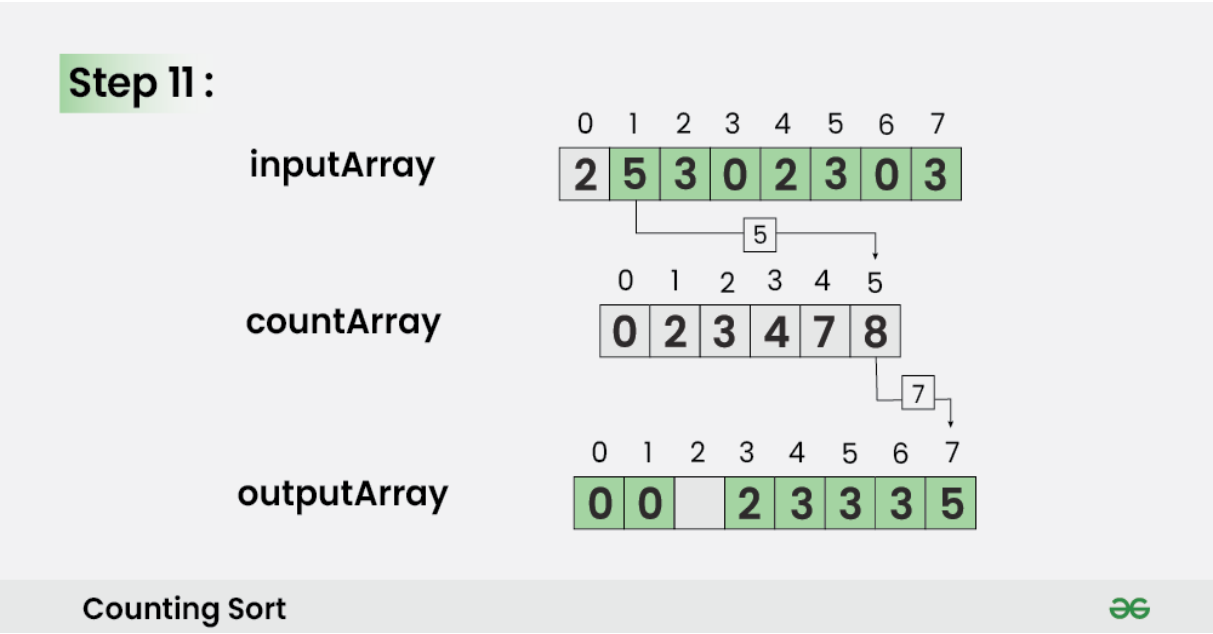
Step 10:

For i = 2, Update outputArray[countArray[inputArray[2]] – 1] = inputArray[2]. Also, update countArray[inputArray[2]] = countArray[inputArray[2]]–.



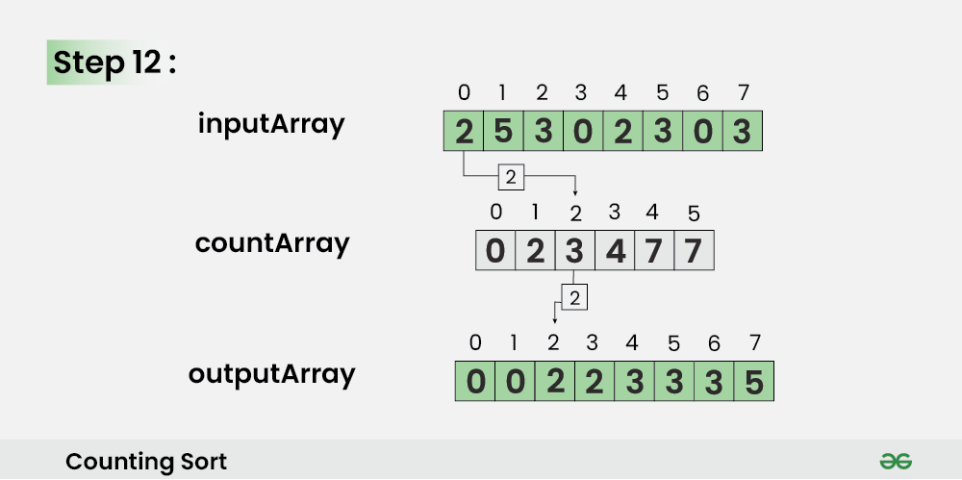
Step 11:

For i = 1, Update outputArray[countArray[inputArray[1]] – 1] = inputArray[1]. Also, update countArray[inputArray[1]] = countArray[inputArray[1]]–.



Step 12:

For i = 0, Update outputArray[countArray[inputArray[0]] – 1] = inputArray[0]. Also, update countArray[inputArray[0]] = countArray[inputArray[0]]–.



Counting Sort Algorithm

- Declare an auxiliary array **countArray[]** of size **max(inputArray[])+1** and initialize it with **0**.
- Traverse array **inputArray[]** and map each element of **inputArray[]** as an index of **countArray[]** array, i.e., execute **countArray[inputArray[i]]++** for **0 <= i < N**.
- Calculate the prefix sum at every index of array **inputArray**[].
- Create an array **outputArray[]** of size **N**.
- Traverse array **inputArray[]** from end and update **outputArray[countArray[inputArray[i]] – 1] = inputArray[i]**. Also, update **countArray[inputArray[i]] = countArray[inputArray[i]]–**.

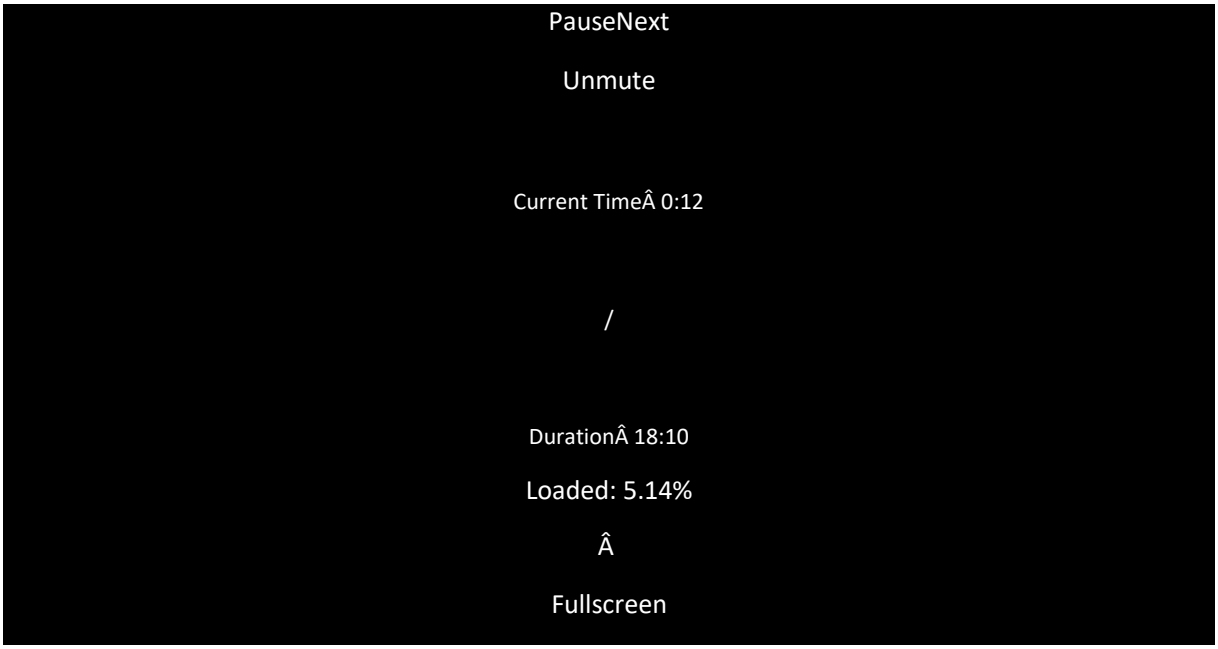
Quick sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.



Algorithm:

```
1. QUICKSORT (array A, int m, int n)
2.  1 if (n > m)
3.  2 then
4.  3 i ← a random index from [m,n]
5.  4 swap A [i] with A[m]
6.  5 o ← PARTITION (A, m, n)
7.  6 QUICKSORT (A, m, o - 1)
8.  7 QUICKSORT (A, o + 1, n)
```

Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

```
1. PARTITION (array A, int m, int n)
2.  1 x ← A[m]
3.  2 o ← m
4.  3 for p ← m + 1 to n
5.  4 do if (A[p] < x)
6.  5 then o ← o + 1
```

7. 6 swap A[o] with A[p]
8. 7 swap A[m] with A[o]
9. 8 **return** o

Example of Quick Sort:

1. 44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than **44** respectively.

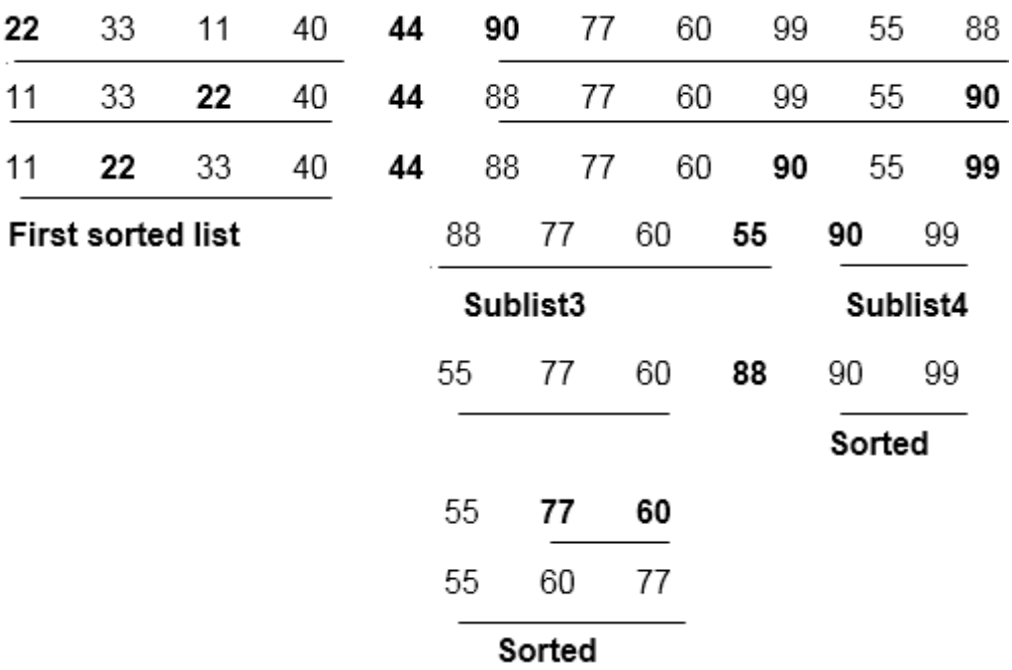
Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 **Sublist2**

And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.



Merging Sublists:

11	22	33	40	44	55	60	77	88	90	99

SORTED LISTS

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.